Wire Cell Toolkit
Architecture and Development Status

Brett Viren

Physics Department

BROOKHAVEN
NATIONAL LABORATORY

Wire Cell Summit 7-9 Dec 2015

Outline

WC Prototype vs. WC Toolkit

make it work	$\rightarrow$	make it fast
initial development	$\rightarrow$	long-term improvements
one developer	$\rightarrow$	many developers

Some commonalities and differences:

- C++11, explicit data models, various I/O
- portable, waf-based build, lives in GitHub.

	Prototype	Toolkit
internal deps	tightly coupled	pervasive use of abstract interfaces
ROOT	intimate	independent (only tests + I/O libraries)
interface	many `main()`'s	API, single, general-purpose CLI
execution	single threaded	abstract "data flow programming" engine
configuration	hard coded	"configurable" interface, JSON files
app construction	hard coded	DFP graph, dynamic plugin system
maintenance	Xin hacking!	long-term, multi-developer
unit tests	some	many
algorithms	state of the art	playing catch-up

Toolkit Design Goals

Want the toolkit to:

- be portable to multiple ('nix) architectures: laptop/workstation, Grid, HPC, including GPU.
- support multiprocessing with (more or less) thread-unaware algorithms.
- dictate interface but not implementation.
- support multiple independent algorithm developers.
- encourage fine-grained unit testing.
- Provide cheap package creation and aggregation.

Toolkit vs. Framework

	Toolkit	Framework
`main()`	✗	✓

Wire Cell Toolkit does:

- dictate transient data model and active class interfaces.
- provide a structure in which to implement functionality.

Wire Cell Toolkit does not:

- determine user interface,
- enforce an execution model,
- nor enforce file formats.

But it does provide some "batteries included" for all of these.

High-level Wire Cell Toolkit Design Concepts

interfaces all toolkit components implement and communicate through abstract base classes.

data model instances are accessed via const shared pointers to their interface class for safe memory management.

compute model units ("nodes") defined as interfaces consuming and producing data model interfaces.

factory concrete interface instance construction via named lookup, supports dynamic plugins.

configurable components may accept parameters from a unified configuration system.

application component aggregation left to developer/user discretion or through toolkit facilities.

Interfaces

Interfaces define the *verbs* and *nouns* of the Wire Cell language.

Data model examples:

- IData transient data base class.
- IWire, ICell defines wire/cell geometry
- IDepo, IDiffusion simulation intermediates.
- IFrame, ITrace defines waveform data

Compute unit examples:

- ICellMaker, IDrifter, IFramer, ICellSelector are examples of transformative nodes (eg, IBufferNode, IFunctionNode).
- IWireSource, IFrameSource are source nodes. and ICellSliceSink sink nodes.

More interfaces exist and more to be added as we progress.

All interface classes go in the package wire-cell-iface.

Named Factory Method and Plugins - using

NamedFactory provides string-to-object lookup supporting configuration and application levels.

```
string cname = "MyClass";
string iname = "my happy instance";
IMyInterface* obj = factory::lookup<IMyInterface>(cname, iname);
```

- Find an instance implementing an interface by its concrete class **name** and an optional instance name.
- Behind the scenes, may use optional plugin system to check shared libraries for `MyClass`.
- Identical lookups return same instance, default-construct if not yet seen.

Named Factory Method and Plugins - Back end

```
WIRECELL_NAMEDFACTORY_BEGIN(BoundCells)
WIRECELL_NAMEDFACTORY_INTERFACE(BoundCells, ICellMaker);
WIRECELL_NAMEDFACTORY_END(BoundCells)
```

- Interface implementation must register with (singleton) factory at file scope to bind concrete class name to the interface it implements.

```
PluginManager& pm = PluginManager::instance();
pm.add("WireCellGen");
```

- Application or configuration layer must register shared libraries holding components with a (singleton) plugin manager.

Configuration

This is still being developed so still partly conceptual:

- Configurable classes implement `IConfigurable` and register with `NamedFactory`.
- User supplies a JSON file containing a dictionary.
  - → keys map to concrete class/instance names.
  - → values follow target-specific schema.
- Parsed and interpreted by a configuration manager.
- Data driven interpretation but some special cases:
  - plugin manager must be configured early
  - execution manager must be instantiated and configured
  - execution manager configuration drives the rest.

This dance must be done at application level but will be presented as a few high-level calls.

Application

It is up to the application developer to determine the scope and structure of how Wire Cell Toolkit components are called.

Many Wire Cell Toolkit applications possible:

- A general-purpose `wire-cell` command line program is being developed and included with the toolkit.
- External framework modules may be created.
- A backend service is being considered in support of Bee 2.0.

Data Flow Programming

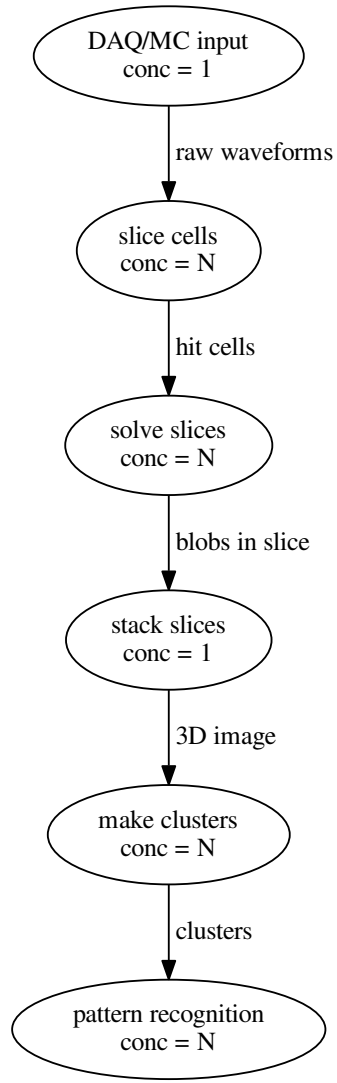
DFP structures the program as a graph.

vertices compute units ("nodes")

edges data queues

- Thread safe queues + stateless nodes = "easy" parallel processing.
- Statefull still possible with concurrency=1 nodes.
- Nodes can be developed and tested in isolation.
- App-level programming by "drawing" the graph.
- Streamed processing can minimize RAM usage.
- Feedback loops may implement iterative flows.
- May instrument graph to collect performance data.

One possible example →



Abstract Execution Model

Wire Cell Toolkit defines abstract nodes and connection method:

- A node has zero or more input/output "ports".
- A port carries data of a specific (data model) type.
- A node declares its maximum concurrency.
- Node and port interface classes.
- "Well known" node types defined as mid-level interface classes.
- IDataFlowGraph implements connection and graph execution methods.
- Battery included: Intel TBB-based implementation.

This works now but needs some final design polish.

Current package set

`wire-cell-`

- `build` default source code aggregation and suite-wide build context.
- `util` NamedFactory, PluginManager, ConfigMangaer, Units, 3D vector, special containers.
- `iface` data model and node interface classes.
- `gen` simulation of 4/6 "D"s: Deposit/Drift/Diffuse/Digitize (Detector response and Deconvolution still in development)
- `alg` reference Wire Cell algorithms ported from prototype.
- `tbb` Intel TBB based DFP execution model implementation.
- `apps` Provided end-user applications.
- `docs` User/developer/installer manual.
- `sst` Celltree file reading/writing.
- `bio` Bee JSON file production.
- `rio` Toolkit ROOT-based I/O.
- `rootvis` ROOT-based visualization.

All are working at some level but are still in development.

A Wire Cell Toolkit Package

Three possible products from building a toolkit package:

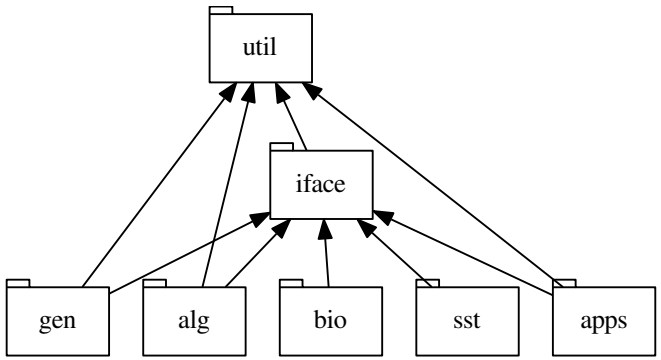
lib `inc/PackageName/*.h` and `src/*.cxx` turned into shared library+headers.

app `apps/main-app.cxx` each source file made into an executable file "`main-app`".

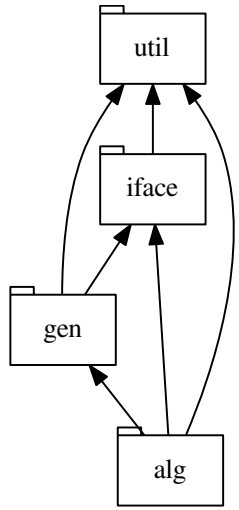
test `test/test_*.cxx` each source file built to a test executable file and run as part of build each time code changes require it.

- Each product type has own dependency tree (see next).
- There is a very low effort barrier to create new packages.
- Consider making a new package before adding to an existing one.
- Also can make new/personal build-aggregation packages to exercise narrower build contexts.

Package dependencies



example of allowed library dependencies



example of allowed
test dependencies

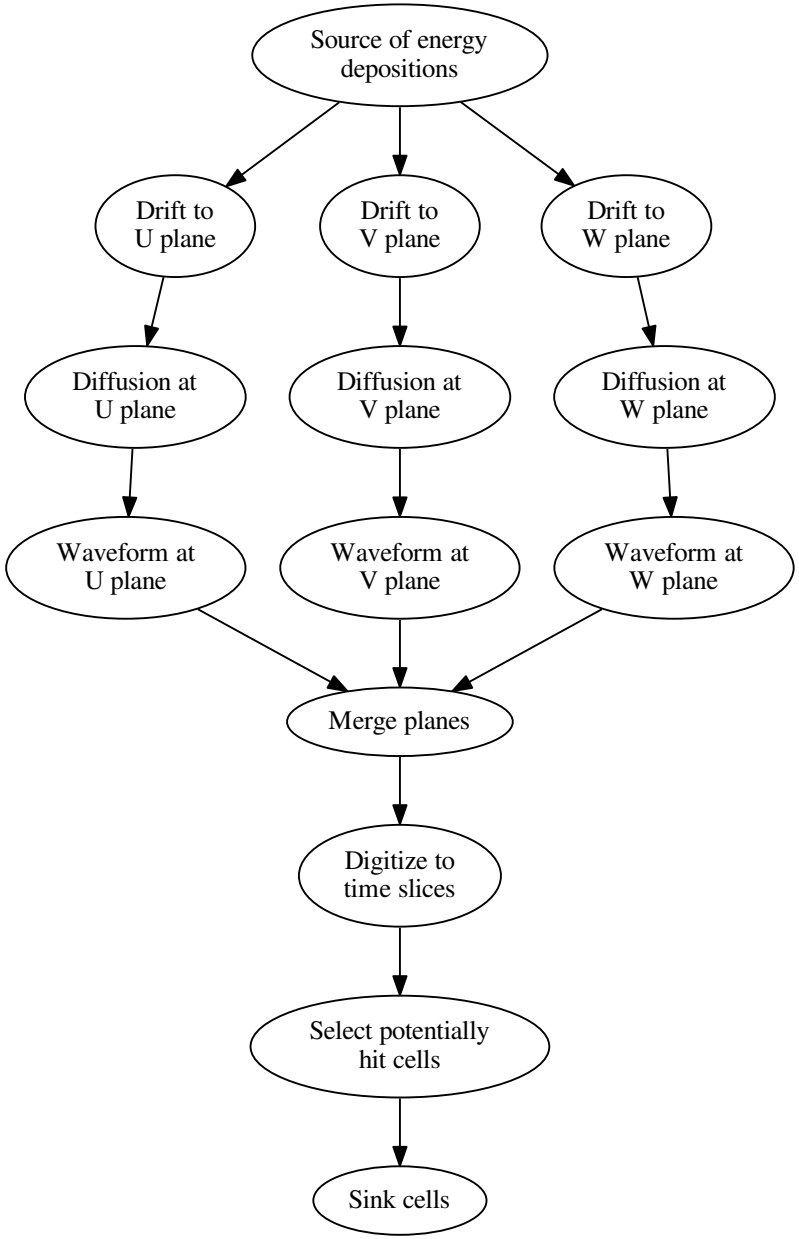
- No direct coupling among implementation libraries allowed, only loose via `iface`.
- `iface` provides "simple" data model implementation.
- Other packages provide data model imp to optimize memory/CPU (eg, lazy instantiating).
- Library and app dependencies strictly controlled, tests may violate.

Hack on your own Wire Cell packages!

1. Make a git repository in GitHub (or wherever you prefer).
2. `mkdir -p inc/MyPackageName src test` and make a `wscript_build` file based on existing ones.
3. Fork wire-cell-build to add your package, or make a personal/reduced equivalent.
4. Implement some existing DFP node interface class or work with me to develop new ones.
5. If your node makes data, use existing "simple data" (eg) data model classes, subclass data model interfaces to implement your own or work with me to extend current data model.
6. Write unit tests as you develop.
7. Run a full-chain application with the `wire-cell` command line program (still in development).

Current Full Chain

- Kinematics are just trivial straight-line "tracks".
- So far, focused mainly on simulation nodes.
  - Still exercises all needed DFP features (sources, sinks, buffering and parallel)
  - But with faster/simpler algorithms.
  - Also for now, short-circuit *detector response* + *deconvolution* steps.
- The only Wire Cell imaging algorithm so far is selecting potentially hit cells.
- Sink node dumps a Bee file.
- wire-cell-tbb/test/test_tbb_dfp.cxx



Current Wire Cell Toolkit Full Chain

Status

- ✓✓✓ repos, build, packaging, dependencies all "done"
  (will evolve as we port the prototype).
- ✓✓✗ initial data model established,
  needs to grow as more algorithms are ported.
- ✓✗✗ only the tiniest, first Wire Cell algorithm ported.
  Lots of work needed here, expect to do tuning/refactoring.
- ✓✓✗ simulation needs 2 more "D"s: detector response and deconvolution.
- ✓✓✗ parallel DFP working, but needs some small design tweaks.
- ✓✗✗ initial end-user configuration, straightforward to flesh out.
- ✓✗✗ celltree, Bee, native I/O needs fleshing out.
- ✓✗✗ general command line app started.
  Waiting on other progress (mostly dfp + cfg).

Contributing to the Toolkit

Tookit development:

- Requires significant learning of current structure, good grasp of OO patterns, understanding OO vs GP paradigms, threading issues.
- Possible overlap with active R&D in the Gaudi world.
- I'd love some help/input here!

Implementing various needed DFP nodes:

- A basic understanding of the toolkit structure required.
- Porting algorithms from the prototype requires reading and understanding Xin's code (nontrivial but not impossible), looking for ways to factor it into well defined DFP nodes joined by well defined data.
- I/O modules need fleshing. They just provide DFP nodes, mostly just a matter of typing in code.

Hacking your own ideas:

- Build the toolkit, start a package or two.
- Test out own interfaces for data and nodes.
- Work with me to incorporate changes.

Summary

- Wire Cell is transitioning from the **very successful working prototype** to a carefully designed toolkit to support long-term contributions from many developers, parallel processing and flexible integration with other systems.
- The toolkit supports the data-flow programming paradigm.
- An initial "full chain" (test) application exercises major toolkit functionality.
- Some structural work on the tookit itself is still needed so API is not yet fully stable, many fine-grained tests are developed.
- Much effort is needed to "port" prototype algorithms.
  $\rightarrow$ contributions from others welcome and needed!